Messaging Framework and Notification / Operation Center

Design Prologue

Operations performed by Meshery are often asynchronous, some taking minutes to complete. Users will fire-and-forget these operations, moving onto other tasks as the operations execute, relying on Meshery to track operation progress and to notify them of operation completion.

Related
- Error Code Reference - https://github.com/layer5io/meshery/issues/2107
- Troubleshooting Guide - https://github.com/layer5io/meshery/issues/2399

Design Goals

Meshery has many loosely coupled components. As a unified system, these components benefit from a common framework for defining and managing the lifecycle of their individual, interrelated messages. This document provides specification for:

1. a Messaging Format and Messaging Framework
    a. Message Classification System
2. a Notification Center (status and health of elements under management)
3. a Operation Center (status and history of operations and workflows)

Design Objectives

The designs in this specification should result in describing a common **message format** and **messaging system** between components, enabling uniform access to:

1. component health
2. operation status
3. policy violation
4. workflow history

Wire format and protocol should be defined.

This specification defines how to provide users with the ability to define and manage the **lifecycle of notifications** for different classes of messages:

1. **Errors** - Error and Remediation
2. **Audit** - Logging and Troubleshooting
3. **Policies** - Validation and Analysis

Messaging System

Using CloudEvents, a single messaging format is defined in which each type of message class can use the same format.

In accordance with the reasoning here, the term *event* will be used to describe what will be transported using messages. "*Events* represent facts and therefore do not include a destination, whereas messages convey intent, transporting data from a source to a given destination." For a glossary of terms used in CloudEvents, see here.

Event format

Event Attributes

See CloudEvents primer.
The format is (mostly) specific to Meshery, but the types are not (for attributes defined in the spec). Attributes tagged with an asterisk (*) are defined in the spec. Meshery specific extension context attributes must follow the CloudEvents naming convention and type-system.

Attribute	Description	Type	Example
id*	Required. The unique id of the event. Format: guid.	string	617850cb-fc5c-4eaa-9706-4ba85068f2fa
source*	Required. source + id must be unique. Format: urn:meshery:component-type:component-name[:component-instance-id] Note how the source includes component type, name, and optionally instance ID Component type and name should be the same as the ones exposed by the ComponentInfo API endpoint (ony for adapters as of this writing) and the same as the ones used in component_info.json used by the error util.	URI-reference	urn:meshery:adapter:consul
specversion*	Required.	string	

type*	Required. One of the Meshery event types, see Event Types	string	error
data*	Optional. Contains the payload, for instance the error event data.	specified by datacontenttype	
datacontenttype*	Optional. A JSON-format event with no datacontenttype is exactly equivalent to one with datacontenttype="application/json"		
dataschema*	Optional.		
time*	Optional. Meshery always sets this. Always use UTC.	timestamp	
subject*	Optional.	string	
correlationid	Optional. Meshery extension context attribute. Attribute that can be used to correlate events, e.g. events correlated to a specific request or operation. For adapters, use operation ID for this. Format: guid	string	4bf0ffea-6fbf-4da8-a9fd-3858f6d0e60c

Event Types

Category	**Type**	**Description**
Audit	audit	
	error	
	log	
Policy	registration	Registering a component in the component registry of an installation / solution.
	health	
	operation	
	policy	

Error	pattern	
	bestpractice	
	validation deployment	

Error Events

Users will run into system and cloud native infrastructure issues. These issues will generate errors. Using Meshery's error codes:

1. Users should be able to easily reference troubleshooting documentation to identify and resolve issues using a unique error code.
2. Every error code belongs to a superset called "class". Every class has a predefined pattern (naming scheme) defined.
3. Every error should have a Probable Cause and Suggested Remediation associated (provided).
1. There is no central single single source of authority for a specific message code. The codes are unique to each component. They may overlap between components.
2. Meshery's components should not reuse messages. Components should emit their own errors, and wrap errors returned by functions from other components or libraries.
3. Self-documenting where possible and as easy to maintain as possible.
4. Every error incident would display an error code along with an error statement.

Event Format

The custom error object that has been planned consists of several attributes that makes the error much informative and yet easier to maintain across projects.

The error struct is defined in MeshKit. Type and name given here are for CloudEvents, and correspond to its naming convention and type-system.

Attribute	**Description**	**Type**
`componenttype`	The type of the component that emits this error event. It is also part of the source URI-reference in the cloud event context, but including it here is practical.	string

`componentname`	The name of the component that emits this error event. It is also part of the source URI-reference in the cloud event context, but including it here is practical.	string
`moniker`	A semi-human readable short key used in descriptive reference to the specific event at-hand.	string
`code`	Unique code identifying a specific error within a specific component.	number
`severity`	Predefined hierarchy, see MeshKit.	string
`shortdescription`	For abbreviated display in the UI. A concise notification with incomplete sentences.	string
`longdescription`	For full display in the UI. An extended explanation. May include a stack trace.	string
`probablecause`	Suggests the likely culprit.	string
`suggestedremediation`	A set of solutions for the user to attempt in order to rectify the situation.	string

Error Codes

The error code is a unique identifier code per component. It carries no semantics.

Code Reuse and Source of Authority

Each component (adapter kuma for instance) is the source of authority of its own error codes. Components should always emit their own errors, and wrap errors returned from functions from other components and libraries.

Implementation

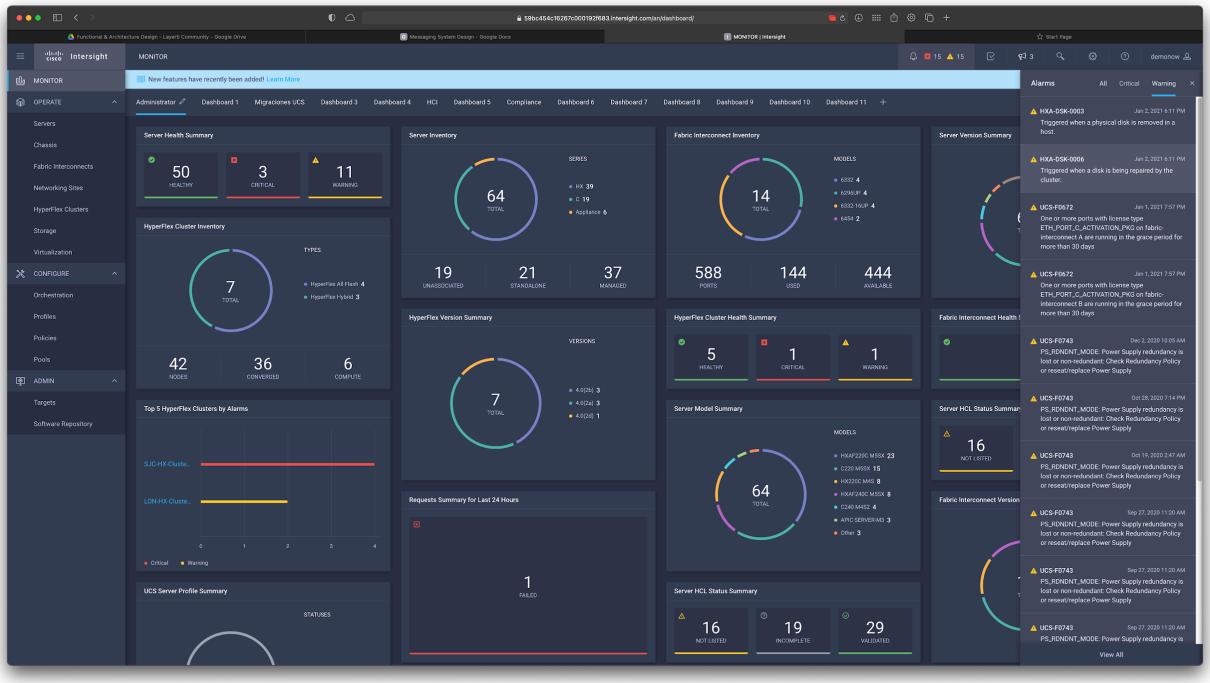
Repository

- config.json file in the root directory containing component type and name.

Go Code

- Each package (that can emit errors) contains an error.go file containing error codes and factory functions for each error.

- Errors are instantiated using e



rrors.New from github.com/layer5io/meshkit/errors.

Tooling

- Verifies that codes are unique within a component (i.e. repository).
- Suggests next free error code(s).
- Possibly updates the code in the errors.New function call if empty/not set.
- Extracts error information (code, cause, remediation) and publishes it.
  Possible solutions:
    - Updates a local md-file
    - Updates a central website.
    - Updates a Google sheet.
- Note that until the tooling is in place, contributors should check manually that codes are not duplicated within a component.

Operation States in adapters-

1. In patternops, currently when the user provisions a cloud native infrastructure or application using a pattern file, no event is streamed back. We need to stream back particular events encompassing the state of that operation that we want to convey.
2. These states can be:
    a. Started provisioning(K8s has been informed)- return a URL with this event. E1
    b. For cloud native infrastructurees-
        i. Control plane in active state E2
    c. For Applications-

i. Application in active state E2

The delta of these states can be calculated by the data from meshsync, where we can use a subscription for a given Operation consisting of n number of states. Where we can define the last state for eg- cloud native infrastructure successfully provisioned and control plane in active state.

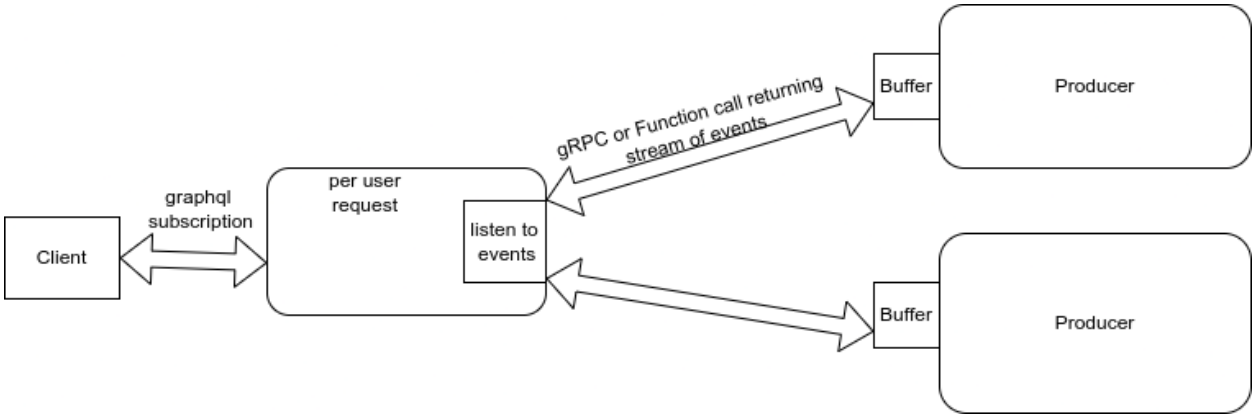
Notification Center

Operations, notifications, events, alarms... all have a lifecycle to manage. The notification center acts as the main user interface from which to do this. Policies that define notifications will be configured in a separate UI.

Meshery's notification area needs to be significantly enhanced.

Figure: See Cisco Intersight for example of messaging and workflows (full-sized image). See Cisco Intersight Demos.

Architecture Diagram



1. Producers are entities which perform operations and produce Events. A `EventBuffer` struct in Meshkit will be used by these entities to create EventBuffer instances.
2. These producers can be adapters performing MeshOpsV1/ PatternOps or can be Meshery server itself. Meshery server will have one instance of this EventBuffer in its handler so that it can produce, buffer and send its own server events like kubernetes components being deployed, etc.
3. A graphql subscription will be created per client. For each graphql subscription, the function will reach out to all connected producers (adapters as well as Meshery's internal producer) and send back all data stored in each buffer. After that, all new events created by producers will be sent back over gRPC/function call.
4. Remove Buffer.
5. Persist events in Meshery

Proposal For Audit and Notification Events [PR]

Broadly the events from the server can be classified as two types logging and notifications, while the logging is for auditing and debugging purposes, the notifications are required for the user instantiated events, the one that is important to been seen and notified to people.

e.g.
Save and Update: Save and update events run continuously for changes done in the design, they can be classified as background events and thus they come under audit events and isn't directly thrown to user with a snackbar. Additionally, the cloud-save icon regularly shows the updates of the design-file, so this is also the cause to add this event as an audit event.

Deploy and Undeploy: Deploy and undeploy events are user instantiated events and they should be notified against their event fired, so this will be classified as a notification and will be sent to user as a snackbar and an entry in the notification success or error panel.

With each event sent from server, there will be a flag that will act as a decision parameter for whether it is classified as audit event or normal notification to be sent in the notification tray with a snackbar.

This PR implements the proposed solution.

Another Approach

suggested by Lee Calcote :
Reference: https://en.wikipedia.org/wiki/Syslog#Severity_level

Each event would come with a severity factor, unlike any additional flag required in the above proposed solution, here the same flag for severity can be used as a deciding factor of whether to classify the event from server as a notification to be sent to the user or as an audit trail.

Value	Severity	Keyword	Deprecated keywords	Description	Condition
0	Emergency	`emerg`	`panic`[9]	System is unusable	A panic condition.[10]

1	Alert	`alert`		Action must be taken immediately	A condition that should be corrected immediately, such as a corrupted system database.[10]
2	Critical	`crit`		Critical conditions	Hard device errors.[10]
3	Warning	`warning`	`warn`[9]	Warning conditions	
4	Error	`error`			
5	Notice	`notice`		Normal but significant conditions	Conditions that are not error conditions, but that may require special handling.[10]
6	Informational	`info`		Informational messages	Confirmation that the program is working as expected.
7	Debug	`debug`		Debug-level messages	Messages that contain information normally of use only when debugging a program.[10]

In that case, the update and save, which is considered as *audit* events would have the severity as “debug”, basically debug notifications are stored in the event trail but restricted to be seen by end users.

Sequence Diagram

<here>

Use Stories

Epic: Transition notifications through their lifecycle

Story 1: Acknowledgement of a Notification

As an operator,
I need to be able to easily acknowledge a notification,
so that others know I have seen this issue and so that we can uphold our customer-facing SLAs.

Implementation:

1.

Acceptance Criteria:

1.

Story 2: Scenario - ...<short title>....

As a [developer/integrator/mesheryctl/operator] user,
I would like to ..... ,
so that .....

Implementation:

1.

Acceptance Criteria:

1.

Appendix

Discussions

Operations Center (August 13th, 2021)
A short demo of Cisco Intersight here -
https://zoom.us/rec/share/9GbIejdcRUwPfKET_fDhIh1c5oiIb003ftx0VaICU2K-L3w8xqAIBlfR8Qky11Hk.DITyOoVzQVJy-B8e

Vijay Cherukuri (Sat. Feb 27th, 2021)
I have a few thoughts on Notifications and also have a few questions. I do not really have a very good grasp of this subject. Here are a few assumptions that I am making. Please correct me if any of the assumptions are incorrect.

1. *Notifications are real-time. Each notification is information about the occurrence of an event. Therefore its value is temporal.*
   [Lee] yes, you're right about the need to be event-driven as the first approach (ideally, in all areas of the architecture). That said, Meshery's notifications will land on a sliding scale of significance (if I may borrow from Twitter: from Fleet to Tweet to Medium Post, so to speak), meaning that each notification will explicitly (ideally) or implicitly carry 1) a severity and 2) have its lifecycle unmanaged or managed. Examples:
   An unmanaged notification: a debug log `mesheryctl system logs`
   A managed notification: a failed provisioning operation or a policy violation
   While the genesis of some notifications will be machine-driven, others will be sourced from a user's action performed.
   [Michael]
2. *Notifications may be meant for the population at large/per group/person.*
   [Lee] Yes, indeed. A subsequent design specification will be needed to address how policies bear weight on notifications and how users, roles, groups intertwine with notifications and policies.
   [Michael]
3. *Because notifications are real-time, they have to be push instead of pull.*
   [Lee] This is largely true, and while there are minor exceptions (ping me for examples), for the purposes of the messaging framework and notification center, we can consider this true.
   [Michael]

Here are some of my questions:

Notifications are being discussed from the standpoint of cloud native infrastructurees. Therefore, there is at least one cloud native infrastructure with one or more microservices. These cloud native infrastructurees preferably would be working silently as one would not want a proliferation of messages. That would be like a dDoS attack. From what I understand, each of those microservices may or may not be from the same vendor. Therefore each of these vendors is at liberty to implement the functionality of the microservice at its own discretion including exception handling. But all that is internal to that microservice. Are there standards on microservices on how to handle errors and how to report them? If it is just a black box, how can one make any assumption about an error code that is returned. It would have meaning only in the microservice. When one categorizes errors into ranges, then one is assuming responsibility for interpreting the error code/message. This would also be a maintenance nightmare as one would need to constantly ensure that the error messages are still being correctly interpreted and that the entire set of messages is accounted for.

Another thing that I wanted to highlight is the idea of a notification center. This requires that the user visit the notification center to view the notification, unless the notification center is invoked each time a notification occurs. This could very well be another nightmare scenario as one would be constantly interrupted. What if the particular device is not available at the time the notification is issued. Should the point at which one receives notification be configurable? Should it be capable of being routed?

Also, the idea of responding to notification. Because cloud native infrastructurees are touted as the means of being able to access functionality from disparate sources thus enhancing sharing, there would presumably be a profusion of microservices. This would, as mentioned earlier, result in innumerable

responses being required, greatly hampering the productivity of the user. The most effective operation would be silent.

Another thing I experienced is that notifications tend to get backed up when the device is offline. Therefore when the device comes online, one is flooded with notifications sometimes making the device inoperable. Moreover, if notifications are considered to be real-time then the backed-up messages may or may not have any utility. Another reason why the notification center (or whatever it is abstracted to) should be routable and configurable.

I do not have the full picture and so some or all of what I put down may be completely irrelevant.

Meeting Minutes

Meshery error code handling #1 (Feb / March 2021)

Participants: Michael Gfeller abishek.kumar@layer5.io

Message

- ✅ Type or class (e.g. error, health, operation, policy, registration) - ✅ in Attributes
- ✅ Message-ID - ✅ in Attributes
- ✅ Sender-ID - ✅ in Attributes
- ❌ Related messages: list of IDs - ❌ No, just use correlation id
- ✅ correlation id: one id to correlated multiple messages, i.e. from an operation that affects multiple components - ✅ in Attributes
- ❌ “requester”: request/requester/session id or similar needs to end up in messages so that the right client/ui picks the asynchronous message up and displays it. e.g. add such a uuid to a ApplyOperation request to an adapter.
  - ❌ No just use correlation id, the requestor should know about this, this could be the operation id in the adapter gRPC call
- Message format version
- ✅ timestamp (utc) - ✅ in Attributes
- if Error message
  - code = component-type.component-name.error-code
    - examples:
      - adapter.istio.1000 (error.go)
      - controllers.meshsync.1000
    - component.json
      - {“type-name”: “adapter”, “name”: “istio”}
  - short desc
  - long desc
  - severity
  - cause
  - remediation
- if health message
  - status

- If operation message
    - Operation-ID
    - Operation-Kind
    - Payload
    - Response-parameters
- If policy message
    - Policy-ID
    - Policy-Format
    - When construct
    - Then construct
- if registration message
    - component-id: e.g. a6aefdc8-6d62-4e8d-bc4c-181ba94f1254
    - component-type, e.g. adapter
    - component-name, e.g. istio

Hierarchies

- Repositories / Projects
- Path (internal_random_test)
- Functionality

Convention:

```
<Component-type.Component-name.Code>
Eg: ADAPTER.KUMA.1000
```

`Docs/ ->` Which would contain other field descriptions

Eg: `ADAPTER.KUMA.1000:`

```
      Cause: “”
      Remediation: “”
      ...
```

Example:

Meeting Minutes

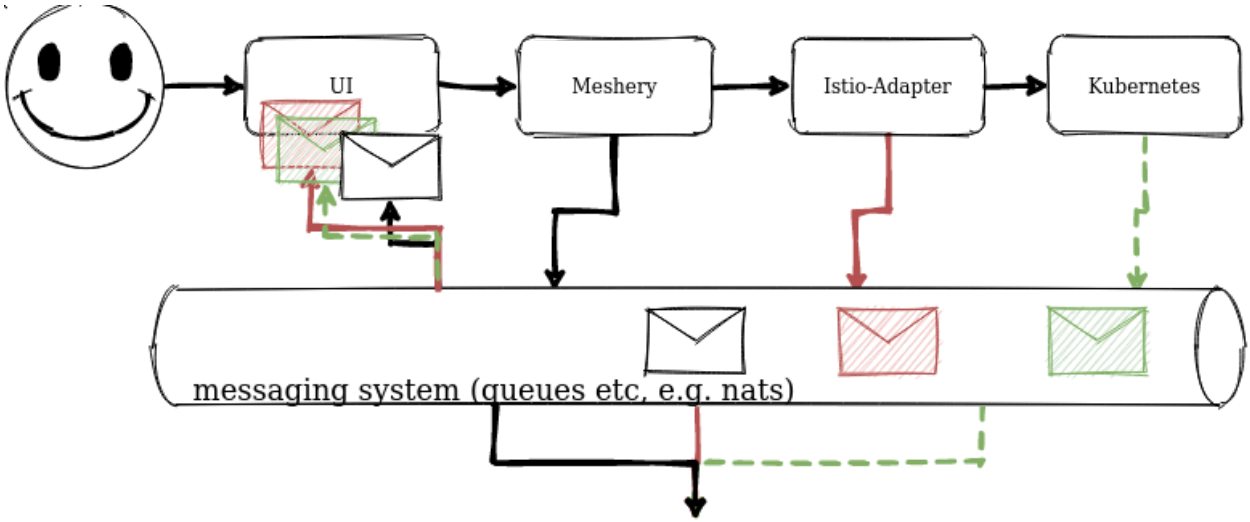
Expanding Notification center (22/08/22) :

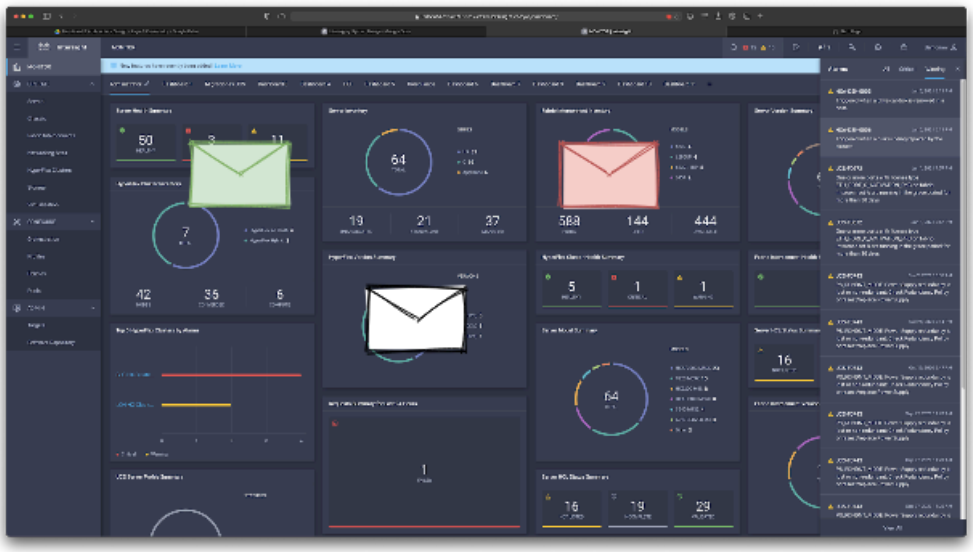
- Currently the UI only sends /theapi/events request when at least one adapter is detected. This should not be the case.
- In next iteration, /api/events should be completely replaced by a graphql subscription
- Details of a notification should have a timestamp

Action Items:

- [Uzair] Merge adapter library changes -> Release -> Meshery PR merge -> Adapters merge

- [Ashish] Events package in meshkit. Adapter library PR merge -> adapter library release -> meshkit changes -> meshkit release -> [Uzair]Meshery changes -> Meshery release
- [Ashish/Uzair] Put in place graphql subscription in place of current logic of /api/events





notification center

```
{
    "class": "error",
    "id": "9f3d831e-0b85-4cf8-9c3b-17e2bac622c3",
    "component-type": "adapter",
    "component-name": "istio",
    "component-id": "a6aefdc8-6d62-4e8d-bc4c-181ba94f1254",
    "correlation-id": "a3a59ca4-72d0-4c01-9f14-fe4be693bdc3",
    "requestor-id": "885edbce-6f15-401c-8c1e-ca7b2611bb8a",
    "timestamp-utc": "2021-02-25T19:40:50Z",
    "error": {
        "code": "1010", // unique because component type, name, and id are part of all notifications
        "short-desc": "error running istioctl command",
        "long-desc": "error running istioctl command: file not found",
        "severity": "ERROR",
        "cause": "istioctl not installed or not in path",
        "remediation": "install istioctl or fix path"
    }
}
```

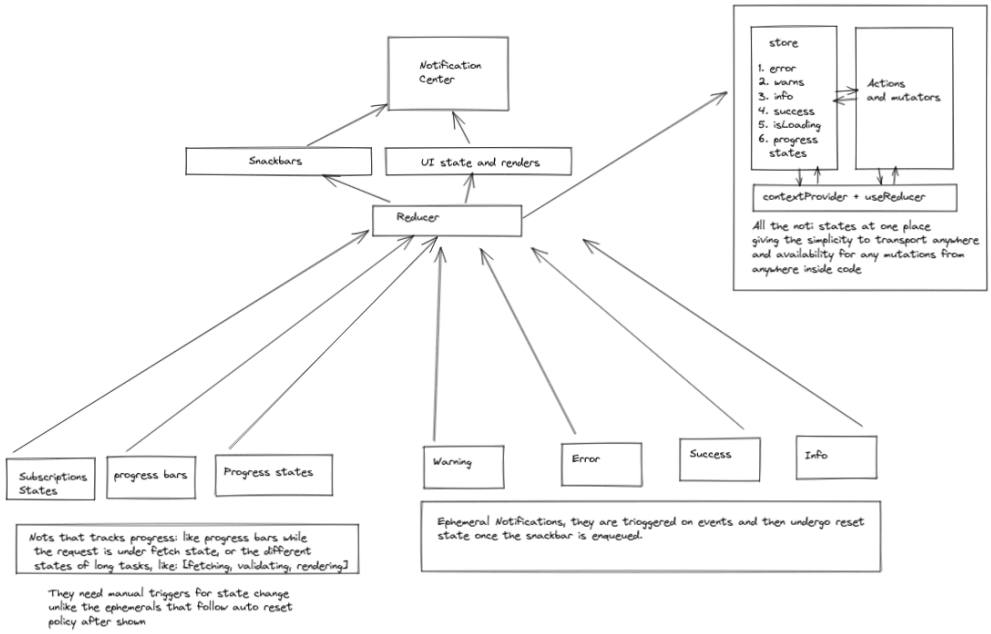
Meshery error code handling 21 (25 March 2021)

Participants: Michael Gfeller abishek.kumar@layer5.io
Conclusions:

- Error codes unique for each component, i.e. no centrally managed error code ranges
- No semantic meaning for specific ranges, as for HTTP codes, due to coordination and maintenance overhead
- No component type and name information in the error message, as this is handled by the cloud event context attributes, specifically the source attribute (see Attributes).
- Implementation:
    - error.New(...) not error.Default(..)
    - a json file in the root of the repo defining component type and name, and min error code (e.g. 1000).
- Tooling:
    - checks that no duplicate error codes are used
    - suggesting next free code
    - possibly inserting code into error.New() if it is empty there - nice to have
    - check notification handler is configured correctly with component type and name, possibly updating it

DRAFT: Meshery Extension (Kanvas)

See design:
https://excalidraw.com/#json=qQtiJePn_eeDELlTn60t9,E_kAxmjkvI6p6GOMDpxo6g



UI:
The Notification Sources:

1. From Server:
    a. The NATS subscriptions that informs about the state of work done in the kubernetes cluster
    b. The gra phql subscriptions that may have been fired from several tasks

2. From Client:
    a. The trigger events that are fired by the user's interaction with the UI.

Handling Notifications:
Since the Notifications panel can be fired and be used from any react component, the Notification State need to be global.

We are using React-Redux for state management, the same could be used for the Notification Management.
But using redux-store can further increase the complexity of actions and stores and high chances of being lost in the code and low scalability.
The solution: The Redux slices:
A Redux Slice is a collection of reducer logic and actions for a single feature in an application. We can use the redux-slice for the notification center and manage the reducers with ease.

The Functionality:

1. The Server Logs:
    a. The server logs whenever comes need to be registered in the Notification center. Once registered, the Notification center can leverage useEffect to catch the state change and show notification inside Notification Menu.
2. The Client Logs:
    a. Creating a dispatch hook that could ease the use of firing events can be leveraged here.
    b. The Notifications triggered by the client can be on a high level of two types:
        i. The Ephemeral Notifications: The Notifications that need to be reset once they are fired.
        ii. The Manual triggers: The notifications that are constantly watching the state of fired events and informing the user at the same time. This manual notifications are triggered and reset manually. They may show the active process, and the a circular progress around the notification icon can be used to show the progress.

The Store:
The Notification Store Object may look like:

```
{
        “Infos”: [],
        “Warnings”: [],
        “Errors”: []
}
```

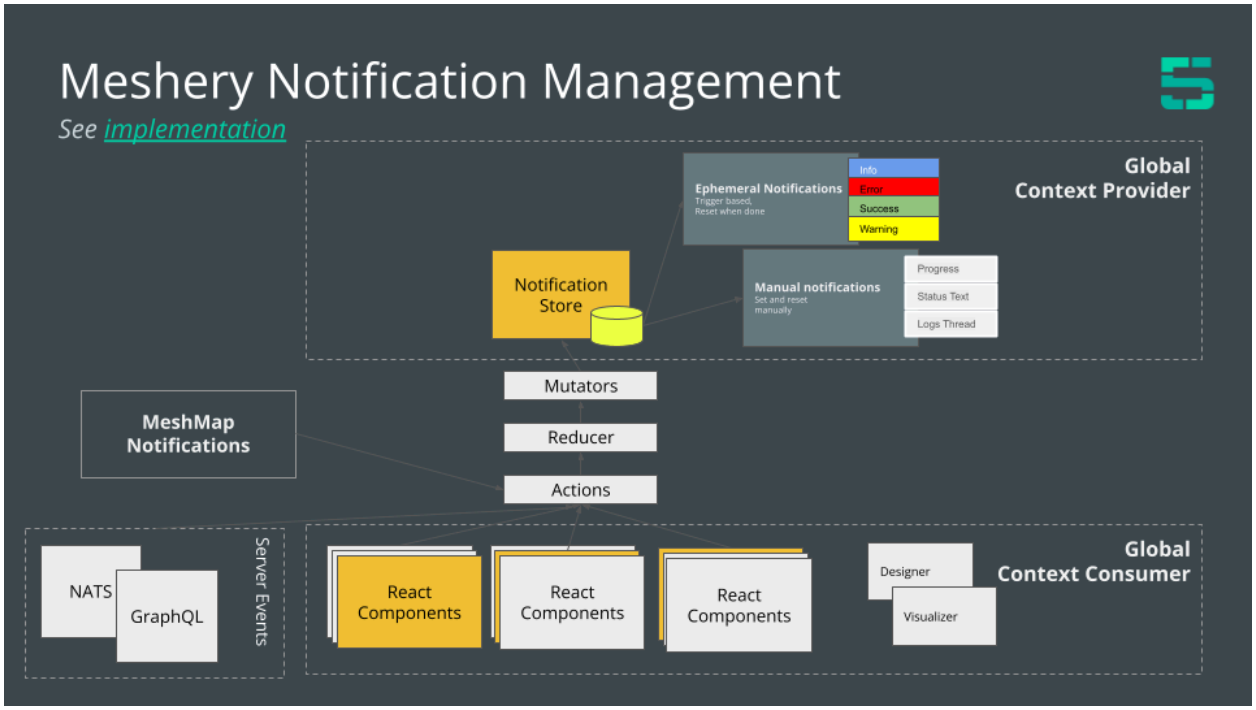
Each holding the further state of notification item like:

```
{
        “Id”: “” ,
        “Summary”: “”,
        “Details”: “”,
        “Cause”: “”,
        “Suggested_remediation”: “”,
        “Severity”: “”,
        “Error Code”: “”,
        “Actions”: []
```

}

Severity:

1. Success: The green snackbar + the green card in the notification panel
2. Info (Meshkit Error: None): The Blue snackbar + the blue card in notification panel
3. Warning (Meshkit Error: Alert): The Yellow snackbar + the yellow card in notification panel
4. Errors: ref
    1. Emergency: The red snackbar + the notification panel red card with the error message and some remediation to get corrected as early as possible. The system may get unusable thus the notification center should force user to follow the remediation steps.
    2. Critical: Red snackbar + red card in notification panel
    3. Fatal: Red Snackbar + red card in notification panel



Steps to take For UI:

1. Create a slice of store for managing Notifications in its own folder/file
2. Create a reducer to take all the action and payload to mutate the react state
3. Create another slice of reducer for managing the extensions notification
4. Create a React-component Wrapper that wraps the global Appjs component which can be used to show any notification as a text/snackbar/component in the ui.
5. The current Notification management UI can be used with minor changes.
6. Create a function that intercepts the NATS event or any event coming from the Server and register automatically inside the UI. The consumer should be created globally to intercept any future event for the freedom to use code/extend anywhere else.

7. Create two handler for managing the manual and ephemeral notifications. The ephemeral notifications should need to be reset with the setTimeout call after their register.
8. Creating a custom hook that exposes the different notification action is a better choice instead of handling it directly with the reducer.
9. Rest of the actions are already done in the current UI.

The Meshery Extension:
The hook created in step 9 can be passed as a prop to the meshery notification that can be used to fire events from extensions to the Meshery Notification Center.

Proposal: Making Meshery Event CloudEvents Compatible

Proposed Message Schema

green means REQUIRED
orange means Exentension

```
{
  "specversion": "<version of cloudevent spec> in use", (type: string)
  "id": "<id of the event>", (type: string)
  "source": "<URI source of the event>", (type: URI) //Can be a URI or URN
  "type": "dot separated reverse-DNS name", (type: string),
  "severitytext":<Defined in the table above>,(type: string)
  "severitynumber":<Defined in the table above>,(type: integer)
  "traceparent":<traceid>-<spanid>,(type string)
  "trace-id":<traceid>,
  "parent-id":<spanid>,
"category":<category of the event>, (type string)
  "datacontenttype": "JSON", (type: string) /should adhere to RFC2046. If
absent, data can be assumed as a JSON. In our use case, we might always default
to JSON and get rid of this field.
  "data": {},(type: Any MIME type, typically JSON for our use cases)
//structure of data will be calculated from dataschema
"dataschema":<>, JSON schema of the data
 "time": <timestamp>, (type string)
}
```

Sample Message

```
{
  "specversion": "1.2",
  "id": "9375a672-4568-4cbd-a9a1-325d47d654eb",
  "source": "meshery-istio",
  "type": "io.meshery.provisioning.istio.virtualservice",
  "severitytext": "Error",
  "severitynumber":3,// less than 6 can be considered notification by client
  "trace-id":d09f5d7e-dabb-4ecb-a5df-81a77f468053",
  "parent-id":s0ghtd7e-dabb-4ecb-a5df-81a77f468054",
  "datacontenttype": "JSON",
  "category":"system",
  "data": {
      "message": "Failed to provision Istio cloud native infrastructure"
      "summary": "Could not provision Istio cloud native infrastructure",
      "details": "XYZ something"
      "error": {
            "probableCause": "something",
            "suggestedRemediation": "something",
            "errorCode": "1000"
       },
  },
 "time": "2022-12-07T12:34:56.789Z",
}

{
  "specversion": "1.2",
  "id": "9375a672-4568-4cbd-a9a1-325d47d654eb",
  "source": "meshery",
  "type": "io.meshery.provisioning.istio.virtualservice",
  "severitytext": "Error",
  "severitynumber":3,// less than 6 can be considered notification by client
  "trace-id":d09f5d7e-dabb-4ecb-a5df-81a77f468053",
  "parent-id":s0ghtd7e-dabb-4ecb-a5df-81a77f468054",
  "datacontenttype": "JSON",
  "data": {
      "message": "Failed to provision Istio cloud native infrastructure"
      "summary": "Could not provision Istio cloud native infrastructure",
      "details": "XYZ something"
      "error": {
            "probableCause": "something",
```

```
            "suggestedRemediation": "something",
            "errorCode": "1000"
        },
  },
 "time": "2022-12-07T12:34:56.789Z",
}

{
  "specversion": "1.2",
  "id": "9375a672-4568-4cbd-a9a1-325d47d654eb",
  "source": "meshery-cloud”,
  "type": "io.meshery.user.signup",
  "severitytext": “Informational”,
  "severitynumber":6,// less than 6 can be considered notification by client
  "trace-id":d09f5d7e-dabb-4ecb-a5df-81a77f468053",
  "parent-id":s0ghtd7e-dabb-4ecb-a5df-81a77f468054",
  "datacontenttype": "JSON",
  "data": {
      "message": "User with username XYZ signed up"
      "summary": "XYZ signed up with Meshery Cloud using google as provider",
      "details": "XYZ something"
      "error": nil,
  },
 "time": "2022-12-07T12:34:56.789Z",
}
```

Behaviors:

- All attributes except for data are part of the event context and are used by middlewares on any routing system to make decisions on that event. In our case, Meshery server is that central fanout system distributing events based on the context.
- **TYPE**: An event type contains metadata of the event carrying info that can be used later for filtering by the client. It follows reverse-DNS name convention and provides a dynamic type-subtype hierarchy.
- **Source**: Source can be a URI or URN uniquely identifying the source of the event. For internal sources, it can be the name of the component like “meshery”, “meshery-istio”. Or for events in remote providers, it can be “meshery.layer5.io” or “staging-meshery.layer5.io”.
- **ID** should be unique per source. Events with the same ID and Source will be considered duplicates
- **Trace ID** is unique and represents the ID of a composite user level operation which might consist (trigger) sequence of other operations. (So basically it’s a root level operation ID). A trace consists of multiple spans and analogously the root operation consists of multiple sub operations creating a tree. Each node represents a sub operation and the root node represents the root level operation. Events can emit from any node of this tree.
    - From each operation:
        - The operation ID will be put into the created Event’s `parent-id`.

- The operation's trace field will be put into the created Event's `trace-id`.
- For external events which do not have an associated root level operation, both these IDs will be null.

- **severity text and severity number:** To filter on Events of different severities, clients can use "severitytext" or "severitynumber" from top level fields. This will return all Events of that severity. The returned event data may or may not consist of an error.
    - Reason to keep severity number: The severity levels are present in decreasing order or urgency as the ***severitynumber*** increases. It allows for easy filtering at client side without like *Get all Events where Event.SeverityNumber < 4.*
    - Reason to keep ***severitytext:*** It allows clients to do exact matching like *if err=="debug"* without maintaining an extra map of severitynumber to text that matches with the creator of the event.

- **Category** of Events:
    1. **System**
    2. **Performance**
    3. **User**

- **data:** The data is just a JSONB and can be of any form.
- **dataschema:** The dataschema will be used to figure out the structure of **data.** Each of the pre-defined data structs will have a schema. If the schema is missing then data is interpreted as a generic key-value map.
    - Some example structures for data:

1. For system events which carry MeshKit errors or general info for notification

```
 "data": {
      "message": "User with username XYZ signed up"
      "summary": "XYZ signed up with Meshery Cloud using google as provider",
      "details": "XYZ something"
      "error": nil,
  }
```

2. For **user(category=user)** events in meshery-cloud:

```
"type": io.meshery.remote.user.designshare
"category": "user"
 "data": {
      "UserID":<>,
      "Email":<>,
      "Provider":<>,
      "First Name":<>,
      "Last Name":<>,
  }
```

Considerations for “Meshery Cloud” Events

Meshery Cloud produces events as well. One of which is currently the category **user/system**. Under this category, we have various types such as: signup, login,publish_results....
These types will be part of the Event context, inside of “type” field, such as `“type”:“io.meshery.remote.user.signup”.`

Data: [This data will be inside JSONB under **Data** and not directly under a column as it can not be pushed inside of context because this is event specific data]

user_id

Examples: (few of the context fields are omitted)

- In case of **publish_results**:

```
“type”: io.meshery.remote.user.publish_results
“category”: “user”
 "data": {
      “userID”:<>,
      “publishID”:<>,
      “publishedProfile”:<>,
  }
```

- In case of **design_share**:

```
“type”: io.meshery.remote.user.designshare
“category”: “user”
 "data": {
      “userID”:<>,
      “peerID”:<>, //ID of the user the design was shared to
      “designID”:<>,
  }
```

- In case of **catalog request**:

```
“type”: io.meshery.remote.user.catalog_request
“category”: “user”
 "data": {
      “userID”:<>,
```

```
      “approvalStatus”:<>,
      “catalogID”:<>,
  }
```

- In case of **meshery_server_registeration**:

```
“type”: io.meshery.remote.user.meshery_server_registration
“category”: “system”
 "data": {
      “instanceID”:<>,
      .
      .
      .
  }
```

The code will be generic enough such that new (cloud event compatible) events can be easily added in Meshery cloud.

Usage of CloudEvents by Keptn

Keptn has multiple services talking to each other through cloudevents. Like meshkit, they also have centrally defined the creation of those events and structure of the Event Data.

Source: Just as specified, source is the source/origin of the event. In their case, it is usually the service name. Similarly for internal components we can have source as “meshery-istio”, “meshery-linkerd”, “meshery”. For external events, it can be the URI.

```
source, _ := url.Parse("approval-service")
```

```
if source == "" {
    source = "https://github.com/keptn/keptn/api"
}
```

Type: As specified in the doc, they use dot separated to conform to “SHOULD be prefixed with a reverse-DNS name”. The actual task is sandwiched between their predefined prefixes and suffixes. Similar proposal I have, we can predefine suffixes and prefixes in meshkit and operations defined in adapter-library/adapter/meshery/meshkit (These defined operations will also be used inside the operations table). An example event type will be <prefix><operation><suffix>. Where prefix=”meshery.event”, suffix=”.provisioning”, operation=”Istio cloud native infrastructure”/(? Or maybe “MeshOps”)

```
const keptnEventTypePrefix = "sh.keptn.event."
const keptnTriggeredEventSuffix = ".triggered"
const keptnStartedEventSuffix = ".started"
const keptnStatusChangedEventSuffix = ".status.changed"
const keptnFinishedEventSuffix = ".finished"
const keptnInvalidatedEventSuffix = ".invalidated"

const keptnContextCEExtension = "shkeptncontext"
const keptnSpecVersionCEExtension = "shkeptnspecversion"
const triggeredIDCEExtension = "triggeredid"
const keptnGitCommitIDCEExtension = "gitcommitid"
```

```
// GetStartedEventType returns for the given task the name of the started event type
func GetStartedEventType(task string) string {
    return keptnEventTypePrefix + task + keptnStartedEventSuffix
}
```

Tracing: They do not use “traceID” or “spanID” as defined in the extension specification in CloudEvents. Their events propagate in such a way that one triggers another. They put the id of parent event in the child event’s “triggeredid” field to find the trace. They do not have the concept of “operation”

Since we have the concept of Operations, I think we should use CloudEvent/OpenTelemetry distributed-tracing extension to trace at an operation level. And not use this field as it does not fit our use case.

```
func (a *ApprovalTriggeredEventHandler) handleApprovalTriggeredEvent(inputEvent keptnv2.ApprovalTriggeredEven
    triggeredID, shkeptncontext string) []cloudevents.Event {
```

```
  outgoingEvents := a.handleApprovalTriggeredEvent(*data, event.Context.GetID(), a.keptn.KeptnContext)
```

Data: They define the structure of data centrally which is specific to their business logic. We can also have a generic EventData struct in meshkit which will also encapsulate meshkit errors (if present). Our Data format is inside of the above proposed message.

```go
// EventData contains mandatory fields of all Keptn CloudEvents
type EventData struct {
    Project string            `json:"project,omitempty"`
    Stage   string            `json:"stage,omitempty"`
    Service string            `json:"service,omitempty"`
    Labels  map[string]string `json:"labels,omitempty"`
    Status  StatusType        `json:"status,omitempty" jsonschema:"enum=succeeded,enum=errored,enum=unknown"`
    Result  ResultType        `json:"result,omitempty" jsonschema:"enum=pass,enum=warning,enum=fail"`
    Message string            `json:"message,omitempty"`
}
```

An example JSON of Kept CloudEvent context (context means everything except data):

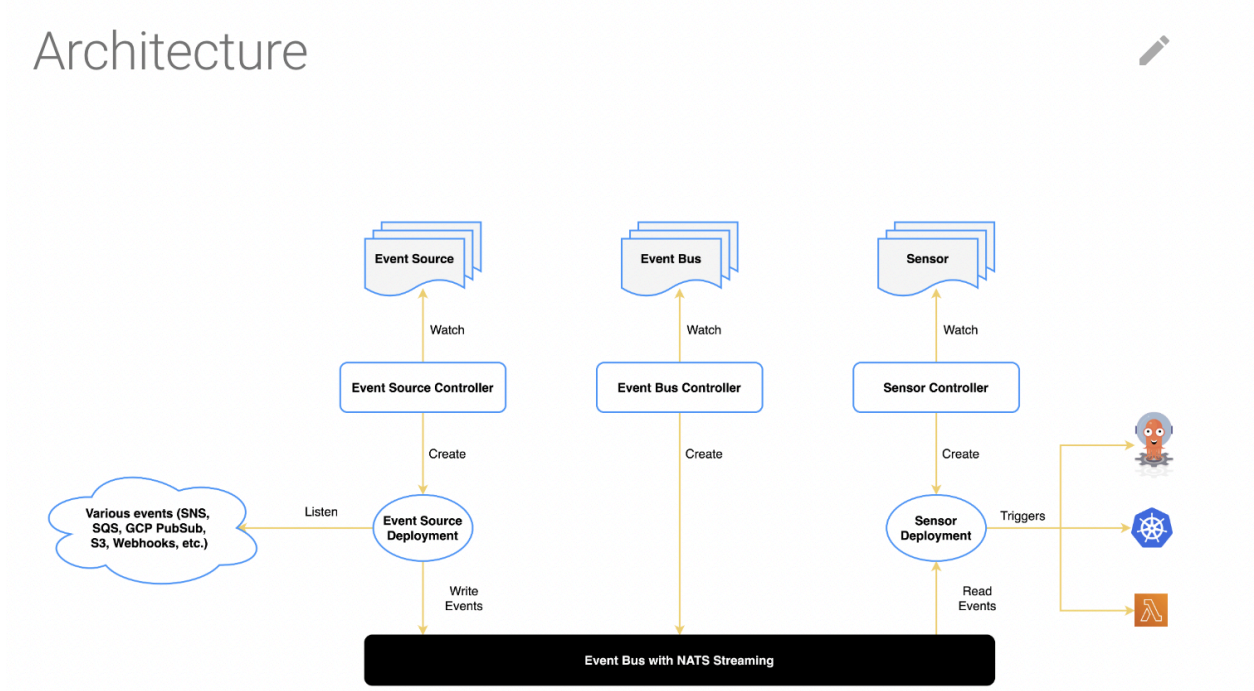
```
    "id": "25ab0f26-e6d8-48d5-a08f-08c8a136a00k",
    "source": "lighthouse-service",
    "specversion": "1.0",
    "time": "2022-02-01T02:46:35.853Z",
    "type": "sh.keptn.event.evaluation.finished",
    "shkeptncontext": "8f884b2a-2197-4e2f-8284-170ea0a66579",
    "shkeptnspecversion": "0.2.3",
    "triggeredid": "991eba72-c520-4da5-ba95-d5876101393c"
```

Argo Cloud Event structure:

```
   {
        "context": {
          "type": "type_of_event_source",
          "specversion": "cloud_events_version",
          "source": "name_of_the_event_source",
          "id": "unique_event_id",
          "time": "event_time",
          "datacontenttype": "type_of_data",
          "subject": "name_of_the_configuration_within_event_source"
```

```
        },
        "data": {
          "header": {},
          "body": {},
        }
    }
```

Argo Architecture:



Action Items:

1. Convert current Event struct to CloudEvent struct internal to Meshery i.e. all produced events inside of Meshery should be CloudEvents and Meshery should be sending out CloudEvents over /api/events
2. Using CloudEvent proto file, migrate StreamEvents RPC to return CloudEvents.

    a. Meshkit will have central functions to create events based on certain options. This can be used by any number of components like Meshery server and adapters.
    b. Meshery server's gRPC client code to receive CloudEvents from Adapters.
    c. Adapter's gRPC server code to send out CloudEvents and use the newly defined StreamEvents RPC.
3. UI using the cloud event js SDK to parse cloud events sent over by Meshery.
4. Add a POST endpoint in Meshery server for /api/events for any external system to be able to POST cloudevents in Meshery server.
5. Meshery Cloud database
6. Mesheryctl considerations